Do code and comments co-evolve? On the relation between source code and comment changes

Fluri, Beat ; Würsch, Michael ; Gall, Harald C

Abstract: Comments are valuable especially for program understanding and maintenance, but do developers comment their code? To which extent do they add comments or adapt them when they evolve the code? We examine the question whether source code and associated comments are really changed together along the evolutionary history of a software system. In this paper, we describe an approach to map code and comments to observe their co-evolution over multiple versions. We investigated three open source systems (i.e., ArgoUML, Azureus, and JDT Core) and describe how comments and code co-evolved over time. Some of our findings show that: 1) newly added code|despite its growth rate|barely gets commented; 2) class and method declarations are commented most frequently but far less, for example, method calls; and 3) that 97% of comment changes are done in the same revision as the associated source code change.

DOI: https://doi.org/10.1109/WCRE.2007.21

Do Code and Comments Co-Evolve?
On the Relation Between Source Code and Comment Changes

Beat Fluri, Michael Würsch, and Harald C. Gall
s.e.a.l. – software architecture and evolution lab
Department of Informatics
University of Zurich, Switzerland
{fluri,wuersch,gall}@ifi.uzh.ch

Abstract

Comments are valuable especially for program understanding and maintenance, but do developers comment their code? To which extent do they add comments or adapt them when they evolve the code? We examine the question whether source code and associated comments are really changed together along the evolutionary history of a software system. In this paper, we describe an approach to map code and comments to observe their co-evolution over multiple versions. We investigated three open source systems (i.e., ArgoUML, Azureus, and JDT Core) and describe how comments and code co-evolved over time. Some of our findings show that: 1) newly added code—despite its growth rate—barely gets commented; 2) class and method declarations are commented most frequently but far less, for example, method calls; and 3) that 97% of comment changes are done in the same revision as the associated source code change.

1. Introduction

"Comment your code!" The task of commenting one's source code is often neglected; even though everybody who is writing software knows the value of good comments [20]. Reading code is a fundamental task during software engineering [10]—and code is read more often than it is written. Even books covering best-practices in commenting exist, *e.g.*, *The Elements of Java Style* by Vermeulen *et al.* [21]. Comments allow one to understand the code faster and deeper and to improve its readability [18, 19]. Especially, they are crucial to sustain software maintainability and aid in reverse engineering, for example when applying the *Read All the Code in One Hour* reengineering pattern [5]. Elshoff and Marcotty already stated in the early eighties that comments as well as the structure of the source code aid in program understanding and therefore reduce maintenance costs [6]. This finding was confirmed by the studies of Ted Tenny [19]. But as the example of Lakhotia shows, sometimes programmers do not care that someone else might want to understand the source code [13].

For maintenance and reverse engineering tasks both, the lack of comments as well as outdated comments are counter-productive. To understand whether the comments are a reason for decreasing maintainability in software projects we address the following research questions in this paper:

1. Does the ratio between comment and source code remain stable over the history of a software project (*i.e.*, is there a recognizable effort made to comment the source code)?
2. Which source code entities are most likely to be commented?
3. Are comments adapted when source code is changed (*i.e.*, are comments kept up-to-date) and when does the adaptation take place—while changing the source code or afterwards?

The contribution of this paper is an approach to map source code entities to comments in the code and a technique to extract comment changes over the history of a software project. We have conducted three experiments on three different open source software systems to answer our research questions: ArgoUML, Azureus, and JDT Core. We describe how comments and code co-evolved over time. Some of our findings show that: 1) newly added code—despite its growth rate—barely gets commented; 2) class and method declarations are commented most frequently but far less, for example, method calls; and 3) that 97% of comment changes are done in the same revision as the associated source code change.

The remainder of the paper is structured as follows. In Section 2 we present our approach to investigate the relation

between source code and comment changes. This approach is then applied to the open source projects in Section 3. We review related work in Section 4 and conclude our findings in Section 5.

2. Source Code Comment Adaptation

Versioning systems such as CVS[1] and SubVersion[2] neither provide features for fine-grained source code change analysis nor for tracking comments. In fact, they are not even capable of providing a mechanism for distinguishing comments from source code. We have overcome this limitation by defining a taxonomy of source code changes and an algorithm to extract them [8]. Our taxonomy defines source code *change types* according to tree edit operations in the abstract syntax tree (AST). Each change type is classified with a *change significance level* that expresses how strong the change may impact other source code entities and whether it is *functionality-modifying* or *-preserving*. We implemented the change extraction as an Eclipse plugin named CHANGEDISTILLER. Our tool uses the release history database approach, similar to the one proposed by Fischer *et al.* in [7], to retrieve evolutionary data of software systems, such as information on different versions including the associated source code. To extract source code changes, our tool pairwise compares ASTs and applies our taxonomy. The algorithm first builds a matching set between AST nodes and then determines the tree edit operations based on the matching set [4]. By using CHANGEDISTILLER, we are able to track fine-grained changes, in the sense that we can detect syntactical changes down to the statement level. For example, we can recognize whether and where a statement was inserted into method `foo()` or that the condition of an `if`-statement of method `bar()` was updated between Revisions 1.1 and 1.2. Moreover, our change detection algorithm also detects comment changes, or more precisely, changes to block, line, and doc comments.

2.1. The Change Detection and Tracking Process

Figure 1 gives an overview on the change detection and tracking process: 1) The source code of all revisions of a particular top-level source code entity (which is a Java class in our current implementation) is fetched from the release history database (RHDB). 2) For each pair of subsequent revisions, we establish a mapping between source code entities and comments. We then extract the change types of both, source code entities and comments. 3) When this process is completed, a full-fledged change history is available for the class, allowing us to relate comment to source code

[1] http://www.nongnu.org/cvs/
[2] http://subversion.tigris.org/

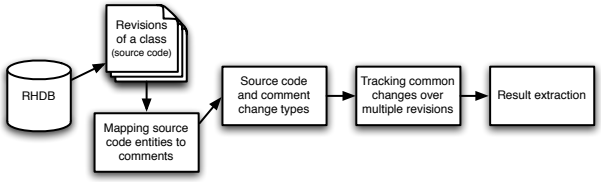


Figure 1. Overview on the change detection and tracking process.

changes and make a variety of observations, ranging from *e.g.*, *The most commented source code entity is...* to more sophisticated ones such as *The comment associated with a particular `if`-statement in method `bar()` was adapted three revisions after the condition of the `if`-statements had been updated.* By aggregating these observations we can especially analyse the comment-to-source-code-adaption-behavior in the investigated software project.

2.2. Mapping Source Code Entities to Comments

In programming languages, it is seldom straight-forward to track relations between comments and source code entities algorithmically. Doc comments, known as Javadoc in Java, are comments that are directly associated with a class, an attribute, or a method and therefore do not need to be mapped any further, whereas block and line comments can not be assigned confidently to a particular adjacent entity by using purely syntactical rules. Because of that, Kaelbling even proposed to remove line and block comments from programming languages and rather introduce *scoped comments* [12]. However, in today's programming languages, we still have to deal with line and block comments and consequently, we have to establish a mapping by applying a set of heuristics.

Since programmers often use consecutive line comments as a syntactical alternative to block comments, line comments are subject to an additional block building algorithm before we establish a mapping between source code entities and comments.

In order to find out whether a comment is associated with his preceding or succeeding source code entity, we apply a set of heuristics:

- **Comment on the same line.** Comments and source code entities, located on the same line, are often associated. These kinds of comments clarify the meaning of the preceding source code entity, as shown in the following example:

```
int i = 0; // Iterator for while loop
```

- **Comment on an adjacent line.** Comments are normally in direct proximity of the corresponding source

code entity, whereas other entities are located less close. In the example below, each of the surrounding statements must be considered to be associated:

```
foo();
/* If foo() did not succeed,
   then calling bar() will
   raise an exception. */
bar();
```

- **Comment describes source code.** Each word appearing in the comment as well as in the source code entity is an indication that the comment belongs to the source code entity. In other words, we use a token-based measure (see Section 2.3.1 for details) to determine the similarity between comment and source code, following the heuristic that comments often pick up, *e.g.*, variable names found in the code that they are describing. Concerning the example above, both, the method invocations `foo()` and `bar()` can be associated to the comment.

For both, the preceding and the succeeding source code entity, we compute a ranking based on these heuristics. We map the higher ranked entity to the comment. In the case that the ranking is even, the succeeding source code entity is chosen, since among developers, it is common practice to write comments preceding the associated source code statement or block.

In the example above, all the heuristics apply on both source code entities `foo()` and `bar()`. They are adjacent to the comment in between them and have the same textual similarity—the words "foo" and "bar" are both in the comment. Since the ranking is even, we choose the succeeding entity, *i.e.*, `bar()` as the belonging entity.

2.3. Extracting Comment Changes

We extended our taxonomy of source code changes with comment change types. For that, we distinguish between changes in doc comments (*e.g.*, Javadoc) and block/line comments. For each comment type, the change types *insert*, *delete*, *move*, and *update* are specified with a name and the corresponding tree edit operation.

By including the comments into the AST, comment changes are extracted and classified. A matching between comments is computed by a token-based string similarity measure.

2.3.1 Token-Based String Similarity

To compare two strings s_1 and s_2 using the token-based string similarity measure, the strings are first split into bags (multisets) of tokens, $T(s_1)$ and $T(s_2)$, according to a given separator. For our concerns, the advantage of using bags instead of sets is that multiple occurrences of the same word are counted multiple times.

The similarity value of the two strings is calculated as

$$sim(s_1, s_2) = \frac{|T(s_1) \cap T(s_2)|}{\max(|T(s_1)|, |T(s_2)|)}$$

We chose the white-space characters as separators to split comments. Source code entities are split with a combination of white-space and '.' for the word matching in Section 2.2. Furthermore, we define that two comments c_1 and c_2 are similar if $sim(c_1, c_2) \geq 0.4$.

2.4. Relating Comment to Source Code Changes

Summarizing the steps described in the previous sections, we have gathered all data that we need in order to investigate whether or not comments are adapted when source code changes: 1) For each comment, we can compute to which source code entity it belongs, *i.e.*, which source code entity it describes; 2) the change types describe when and how source entities as well as comments have changed.

In particular, by combining 1) and 2), we can address:

1. Whether a comment and its belonging source code entity have changed at the same time or the comment changed later,
2. Whether the changes were of the same type (insert, delete, move, or update), and
3. Which source code change type is most likely to trigger a comment adaptation.

Consider the example chain of comment changes in Figure 2. In Revision 1.2, a comment, `/* threshold at 0.8 */`, is inserted for the source code entity (variable declaration) `double t = 0.8;`. The source code entity changes in Revision 1.3, but the corresponding comment is not updated until Revision 1.4. Both, comment and belonging source code entity, are deleted in Revision 1.5.

We reconstruct such chains backwards by starting with the latest revision r_i. For each comment change $c_k \in r_i$ we look whether the belonging source code entity was also changed. If the belonging entity changed as well, we are done and store that there was a common change between the comment and its belonging entity, whether they changed the same way (*i.e.*, insert, delete, move, or update), and the change type of the belonging entity. In our example in Figure 2, we start with Revision 1.5 and the comment deletion. The belonging entity and the comment changed in the same revision and in the same way.

If the belonging entity did not change in r_i, we look for corresponding changes in r_{i-1}, thus go backwards. This

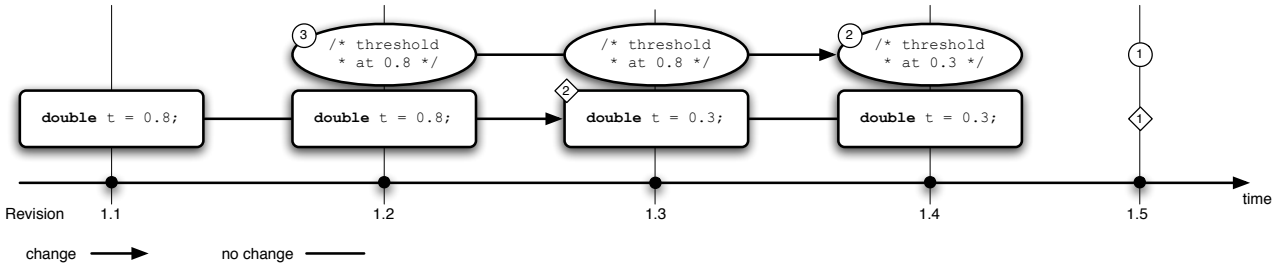


Figure 2. An example of a chain of comment changes; a numbered circle indicates a comment change, a numbered diamond a source code change. Common source code and comment changes have the same number.

step is repeated until we either find a change of the belonging entity, or another change of the comment $c \in c_k$. If another change of $c \in c_k$ occurs, a new element in the chain begins, and we state that c_i occurred without a source code change. In our example, in Revision 1.4, the comment was changed one revision later than its belonging entity. The comment insert in Revision 1.2 happened without a corresponding source code change.

The investigation of our example chain answers the research questions we posed in Section 1 and its results can be summarized for the example as follows: The comment changed three times. The first change (in Revision 1.5) happened in the same revision accompanied by a change of the belonging entity of the comment. Both, comment and entity, changed the same way (delete). The second change (in Revision 1.4) occurred one revision later than the change of its belonging entity, thus, the comment change was triggered by an earlier source code change. The third comment change (in Revision 1.2) was applied solely. We can also state, that it is more likely that a statement delete triggers a comment change in the same revision than a statement update does.

2.5. Limitations

Due to an implementation issue, we are currently not able to establish a proper mapping whenever a line comment follows a block comment directly and vice versa. In addition, issues with successive comments that are in different scopes persist. Under these circumstances, comments are related to comments, and comment changes to comment changes instead of source code changes.

A general limitation concerns source code that is commented out; it is a common practice among developers to let obsolete or debugging statements remain inside of the code by just marking them as comments [18]. Currently, there is no lightweight approach to distinguish these artifacts from *real* comments.

3. Case Studies

We conducted three case studies with open-source software projects to investigate the relation of comment changes to source code. We chose projects of different domains: 1) ArgoUML,[3] a UML modelling tool; 2) Azureus,[4] a Bittorrent client; and 3) Java Development Tools core plugin (JDT Core)[5] of Eclipse. All three projects are written in Java and are version-controlled using CVS.[6] The dimensions of the projects are summarized in Table 1. The number of Java classes (NOC) and the number of lines of code (LOC) are given for the first and last major releases.

Project	Observ. Period	NOC		kLOC	
		first	last	first	last
ArgoUML	Jan'98–Dec'05	1501	1526	87	154
Azureus	July'03–May'07	187	2409	25	312
JDT Core	June'01–May'07	814	1112	188	342

Table 1. Case studies dimensions

Next, we address the research questions that we have posed in Section 1 and expect to gain the following insights:

1. The ratio between comment and source code reveals whether the developers are commenting their code or not. With a trend analysis over the major releases we will see whether the developers are getting lazy or motivated commenting the code.

2. Intuitively, building blocks of source code should have a preliminary comment. We expect, that 1) attribute, class, and method declarations are commented (*e.g.*, with Javadoc); 2) control structures and loops are introduced with a comment; and 3) partly method calls and variable declarations are clarified by comments.

[3] http://www.argouml.org
[4] http://azureus.sourceforge.net
[5] http://www.eclipse.org/jdt/core
[6] ArgoUML has already moved to Subversion. We used the CVS repository provided by the MSR Workshop Challenge of 2006.

3. Keeping comments up-to-date aids in comprehending source code as well as ease the access to the source code for new developers.

In order to answer each research question, we perform a corresponding experiment on the case studies and discuss our findings after each experiment.

3.1. Experiment 1: Ratio between Comment and Source Code

We have selected a set of major releases for each project (ArgoUML 13, Azureus 12, JDT Core 14). For each release, we calculate the number of lines of code (LOC), number of non-commented lines of code (NCLOC), and the number of comment lines (NCL).[7]

In Figure 3 (a)–(c) NCLOC, NCL (left y-axes), and the ratio of NCL to NCLOC (right y-axes) are depicted for each selected release (x-axes) of the three projects.

ArgoUML. The size measurement of ArgoUML is depicted in Figure 3 (a). During the observation period (time between Release 0.9.6 and 0.20a) the amount of source code increased slightly (71k–94k), but the comments almost tripled in size. Since the amount of comments increased stronger than the amount of source code, the ratio between NCL and NCLOC reaches almost 70%, *i.e.*, only a third of the lines of code is not commented. We may state that ArgoUML matured in this period of time and the developers were cleaning and commenting the code base. On the other hand, taking into account that seven years elapsed between Release 0.9.6 and 0.20a, one might also argue, that new source code was introduced and the old one was commented. This might also explain the striking increase of the NCL to NCLOC ratio. With the experiment in Section 3.2, we will get more detailed results to reveal the reasons of the significant increase of the ratio.

Azureus. Comparing the metric values of Azureus in Figure 3 (b) to the ones of ArgoUML, we observe that they evolved contrarily. The code base of Azureus increased tremendously—it increased ten times—whereas the amount of comments only increased slightly. Because of this discrepancy in the growth of NCL and NCLOC, the ratio between them decreased significantly. Azureus seems to be a good (or bad) example for the intuition that the will to put effort in commenting a program decreases over time.

JDT Core. As we can see in Figure 3 (c), NCLOC and NCL grew constantly in the last six year of the project, but NCLOC grew faster than NCL. Interestingly, there is a peak for the first three 3.0.x releases. Between the last 2.1.x and the first 3.0.x release the ratio between NCL and NCLOC increased for about 5%. After Release 3.0.2, the ratio decreased steadily. The possible reasons for the striking increase of the ratio for Release 3.0 are twofold: First, one year elapsed between Release 2.1 and 3.0. In this time, many new features were added to Eclipse. Second, the Eclipse community was growing and one may have wanted to advertise the full-fledged integrated development environment. Both reasons may have had a positive and motivating impact for writing comments.

Between Release 3.0 and 3.1 another year elapsed, but we cannot observe the same effect as in the transition from Release 2.1 to 3.0—rather the opposite. After Release 3.0.2, the ratio between NCL and NCLOC steadily decreased until it fell below the initial ratio in Release 2.0. If we assume that existing comments are not deleted, we can say that the introduction of new source code does not entail the introduction of new comments—although the source code increased strongly after Release 3.1, comments did only marginally. So far, we do not know the reasons why comments are missing.

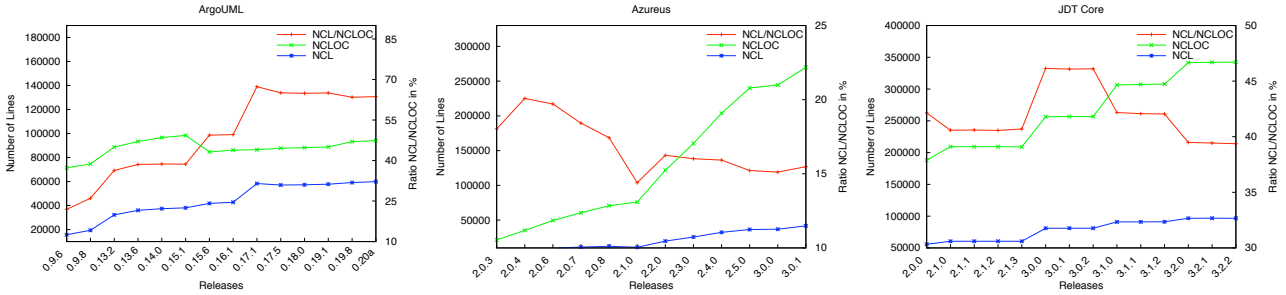
Findings. Newly added code is barely commented. If the code base is increasing linearly—even exponentially for Azureus—, the number of comment lines grew only marginally over a time period up to seven years. Indeed, for JDT Core, the ratio between NCL and NCLOC has a peak around the 3.x releases, but the temporal commenting euphoria decreased significantly in the latest six releases. However, ArgoUML is an outlier: its ratio between NCL and NCLOC increased by almost 40%, meaning that newly added code get commented.

3.2. Experiment 2: Commented Source Code Entities

In the second experiment, we investigate which types of source code entities are most likely to have comments. In addition, we show the proportion of commented entities for each type of source code entity in the projects. For all three projects, approximately 70%[8] of all comments are mapped to the following seven types of source code entity: 1) attribute declaration, 2) class declaration, 3) method declaration, 4) control structure (if- and switch-case-statement), 5) loop (for-, while-, and do-loop), 6) method call, and 7) variable declaration. The remaining 30% of comments are mapped to other source code entities, such as package declarations, import declarations, or try-catch blocks. We expected the seven types to be commented due to the following reasons: First, it is good practice to comment declarations with Javadoc and second, building blocks such as

[7] We used Borland® Together® 2006 Release 2 to compute these metrics.

[8] ArgoUML: 71.5%; Azureus: 61.5%; JDT Core: 75.3%



(a) (b) (c)

Figure 3. Number of lines of code (LOC), number of non-commented lines of code (NCLOC), number of comment lines (NCL), and the ratio between NCL and NCLOC for each investigated project.

control structures and loops as well as special method calls or variable declarations are commented with line or block comments. For the remaining discussion of the experiment we use these seven types.

In Figures 4–6, we display two diagrams for each project. The bar-chart-diagrams show for each release (x-axes) the amount of commented types of source code entities (y-axes)—each type has its own color. The line-diagrams show for each release (x-axes) how often a particular type is commented in percent (y-axes).

ArgoUML. As we can see in Figure 4, the amount of commented class and method declarations increased over the observation period. Except for loops, the other inspected types of entities remained stable. Although the amount of commented attribute declarations stayed stable, their percentage doubled during the 14 releases. This increase is based on the decrease of the total amount of attribute declarations. The amount and coverage of commented method as well as of class declarations increased substantially. For declaration comments (*i.e.*, Javadoc), the percentage of commented declarations increased over time. Since the comments on the statement level were also increased during the 14 releases, we negate the assumption made in Section 3.1 that existing code was uncommented, which led to an increase of the comment ratio. The results from this experiment rather show that the developers of ArgoUML are aware of the importance of writing comments and therefore steadily increase their amount.

Azureus. During the twelve observed releases of Azureus, the amount of all commented declaration and types of statements under investigation increased, as depicted in Figure 5. The percentage of commented attribute declaration stayed stable, whereas for class and method declarations, they decreased. Comments for loops and control structures slightly increased, but for method calls

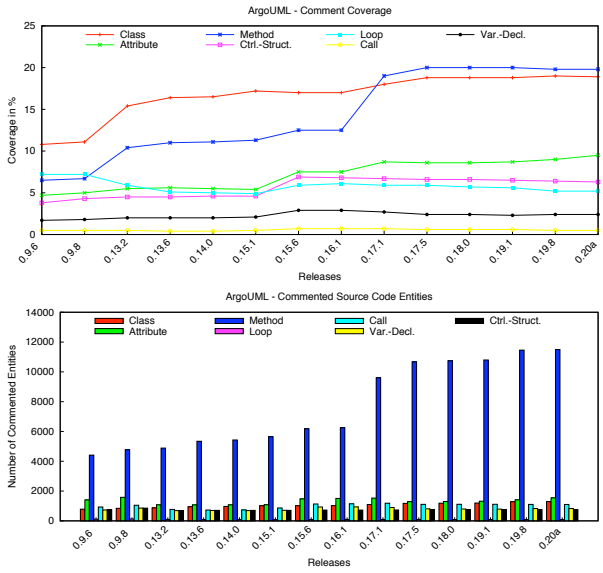


Figure 4. Amount and percentage of commented types of source code entities of ArgoUML.

and variable declaration they remained stable.

Compared to the results of the NCL analysis in Section 3.1, the ratio between NCL and NCLOC falls off more drastically than the percentage of commented source code entities discussed above. Since the former analysis focuses on a line count measure and the latter on comment blocks, such a discrepancy is not surprising. Deleting a comment that spans multiple lines has a stronger impact on the line metric than on the amount of commented source code entities.

JDT Core. In Figure 6, we can observe that the number of commented source code entities augmented for each type

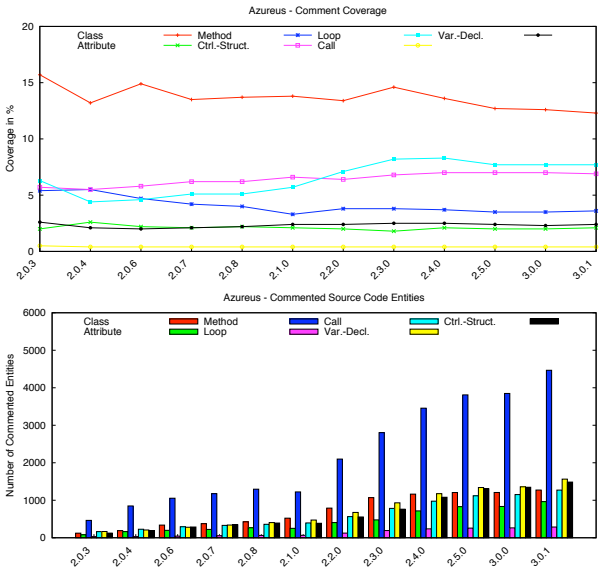


Figure 5. Amount and percentage of commented types of source code entities of Azureus.

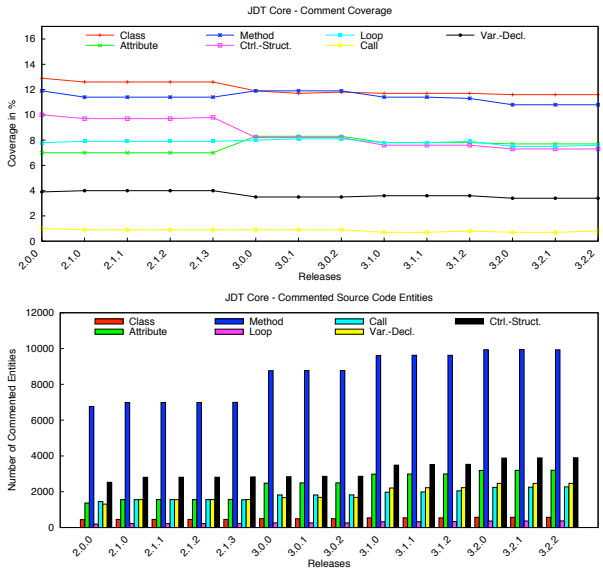


Figure 6. Amount and percentage of commented types of source code entities of JDT Core.

of entities. This matches with the NCL curve in Figure 3 (c) which steadily increased. As the NCLOC increased stronger than the NCL, the percentage of commented class and method declarations as well as of control structures decreased. Except for the attribute declarations, the percentage of the other entity types did not change.

Findings. Class and method declarations are commented most frequently, whereas method calls are scarcely. Again, one project does not comply with these findings: In Azureus the percentage of commented method declarations is not significantly higher than of other commented source code entities.

3.3. Experiment 3: Keeping Comments Up-to-Date

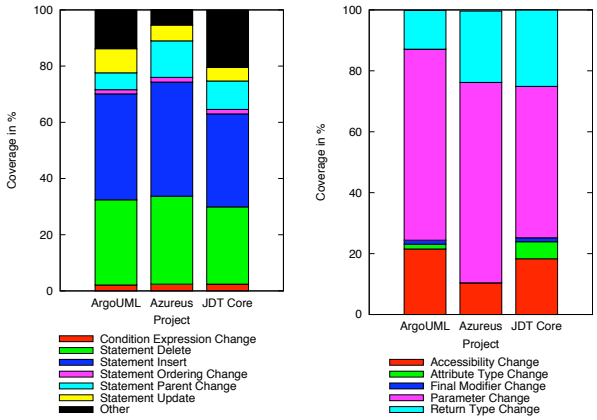
In the third experiment, we reconstruct the chains of all comment changes. The primary goal of this analysis is to find out whether comments are kept up-to-date with the source code. The results of the experiments will show: 1) which proportion of comment changes are due to source code changes, 2) whether the comment and the corresponding source code changes were made in the same revision or the source code change took place earlier, and 3) which source code change type most likely induced a comment change.

ArgoUML. In ArgoUML 35% of all comment changes were line/block comment changes, 65% were Javadoc changes. Of these comment changes, only 23% were due to source code changes, but in 96% of these cases, the comment was changed in the same revision as its belonging source code entity. Surprisingly, 26% of all Javadoc changes were triggered by declaration changes, and 10% only by changes in the method body without a declaration changes—meaning that less than 50% of all Javadoc changes were due to source code changes. We mention possible reasons in the discussion section.

Figure 7 (a) depicts the distribution of the change types in the method body that triggered a comment change. As we can see, the statement inserts and deletes induced the most comment changes. It is also striking that comments are seldom updated because of statement updates.

In Figure 7 (b), we show the change types in attribute, class, and method declarations that were responsible for the occurrence of comment changes. Since parameters of a method and the description of their return values are listed in the Javadoc, it is obviously that Javadoc changes are mostly due to parameter and return type changes. Final and accessibility modifier changes have rarely a changing effect on Javadoc.

Azureus. In Azureus block and line comments (86%) are much more often changed than Javadoc (14%). The percentage of comments that are induced by source code changes in Azureus (52%) is even twice as high as in ArgoUML. 97% of the induced comment changes occurred in the same revision as the source code changes—similar



(a) (b)

Figure 7. Distribution of change types inducing comment changes in the method body (a) and in declarations (b).

to ArgoUML. Almost 50% of all Javadoc changes were made along with source code changes; 25% with declaration changes, 24% by changes in the method body only.

As we can see in Figure 7 (a), the ratio between the change types is similar to the one of ArgoUML. For the declaration changes it is different. In Azureus, neither attribute type changes nor final modifier changes triggered changes in the Javadoc. Accessibility changes are also less frequent, whereas the return type changes have a higher proportion as the accessibility changes.

JDT Core. The results for JDT Core do not show any striking differences compared to ArgoUML and Azureus. Of all comment changes 43% were due to source code changes. Allmost all of them (98%) occurred in the same revision as the change of their belonging source code entities. The comment changes concern 75% block/line comments and 25% Javadoc, of which only 12% were due to declaration changes and 20% of method body changes.

Concerning Figure 7 the differences between JDT Core and the other projects are the higher percentage of "Other" change types and the lesser percentage of parameter changes. We explain and discuss the meaning of "Other" change types in Section 3.4.

Findings. The percentages of comment changes that are due to source code changes are significantly different in the three projects (ArgoUML 23%; Azureus 52%; JDT Core 43%). In particular it strikes, that in ArgoUML 23% of the comment changes are triggered by source code changes, that is only half of the amount of Azureus and JDT Core. After an initial investigation of a number of source code samples in ArgoUML, we found out that in an early revision, around Revision 1.5, the structure of the source code (*i.e.*, indentation) of each class was changed. Since we treat Javadoc as a block-text including white-spaces and we have not yet adjusted our token-based string similarity measure to all occurring special cases, more Javadoc updates were found than one might see by inspecting the code by hand. Because the changes occurred in an early revision, source code changes among the Javadoc changes were missing.

In each of the three projects, the proportion of Javadoc changes that were induced by either declaration changes or method body changes is rather low: ArgoUML 39%, Azureus 48%, JDT Core 32%. Besides the missing tuning of the similarity measure for Javadoc changes, another possible reason we are currently investigating comes into play. Addition, deletion, and updates of Javadoc in unchanged interfaces can never have corresponding source code changes. Assume an interface without any Javadoc is checked into CVS in the Revision 1.1. Its source code was not changed for Revision 1.2, but each method declaration was commented with a Javadoc. These inserts are not triggered by any source code changes, since no changes are recorded for Revision 1.1.

Surprisingly, of all comment changes triggered by source code changes about 97% are done in the same revision as the source code change. This is interesting, since we did not expect that developers change source code and comments together.

3.4. Threats to Validity

To address the validity of our experiments we have to consider three issues. First, the accuracy of the change extraction is closely related to the performance of the matching algorithm. The basic tree edit operations are calculated according to a matching set. A non-optimal matching set generates too many edit operations, *i.e.*, the set of operations is larger than needed to transform a tree into another. However, the transformation is always correct. To evaluate the accuracy of our matching algorithm, we have conducted a benchmark with over a thousand changes [9].

Second, we checked a reasonable set of mapping pairs, but an elaborate qualitative evaluation of the algorithm for mapping of source code entities to comments is still needed in order to generalize its applicability.

Third, as we stated in Section 2.5, a number of comments are related to other comments instead of source code entities. This drawback results in comment changes that are due to comment changes. The category "Other" in Figure 7 (a) shows the percentage of such comment changes. In detail, these proportions are 12% for ArgoUML, 7% for Azureus, and 20% for JDT Core. Since we count for each comment change the change of its belonging source code entity, changes between comments are counted twice. This

results in an error rate between 4% and 10% which we consider acceptable for such an experiment. However, we are investigating possible solutions for such mismatches.

3.5. Résumé

In this section, we reflect our findings to our expectations at the beginning of this section. We split the discussion into three parts:

Ratio between comment and source code. The ratio between the source code and comments evolves differently in the three projects. In ArgoUML the ratio increased, whereas it decreased in Azureus and JDT Core, except for the peak in JDT Core's Releases 3.0.x. We can state that the effort taken to comment the code is higher in ArgoUML and JDT Core than in Azureus. However, this statement has to be checked with an in-depth analysis of the comments to exclude source code that is commented out.

Commented source code entity types. As we have observed in the second experiment, the commented types of source code entities confirm our intuition: Javadoc for declarations, control and loop structures, as well as special method calls and variable declarations are commented most frequently. But the trend analysis showed that the amount of commented entities is neither stable nor increasing. However, in that experiment we did not distinguish public from private attributes, class, and methods; thus, the we cannot state for sure whether the public APIs become less commented or not.

Comment kept up-to-date. Prior to conducting this experiment, we expected that more comment changes are due to source code changes than we have found. We anticipated with at least 50%, but the results of two projects were below (ArgoUML 23%, JDT Core 43%). In the previous section, we have already discussed a possible explanation, but we intend to further investigate these findings with more case studies and in-depth analysis of the existing results. Moreover, the fact that over 97% of all common changes between source code and comments were in the same revision did also surprise us. We did not expect that developers are that disciplined. On the other hand, with our approach we can hardly tell whether the reason for a comment change was due to a source code change, because we did not (yet) investigate the semantics of comments and their changes.

We have seen that for Javadoc changes, return type and parameter changes are most influencing. This is beneficial as they directly impact the correctness of APIs.

Eventually, we can say that our studies gave a first insight into the comment change behavior of open-source software. The three chosen projects are in different domains, are well-known, heavily used, and still under development. Nevertheless, in order to make a more general conclusion, we have to analyse further projects and tune our approach.

4. Related Work

In [11], Jiang and Hassan conducted a study on the evolution of comments in PostgreSQL. They investigated how many header comments and non-header comments were added or removed to PostgreSQL over time. In contrast to their work, we do not restrict ourselves on studying the addition and deletion of comments, but also track updates and moves. Moreover, we integrate source code change analysis down to the statement level in order to track whether and how source code and comments change together.

Antoniol *et al.* proposed a method based on information retrieval to recover traceability links between source code and free text documents [1]. Marcus and Maletic propose a similar solution in [16]. However, both approaches focus on external documentation and do not investigate evolutionary aspects, *i.e.*, they do not track documentation and source code changes together over time. Recently Witte *et al.* used Semantic Web Technologies connect software and documentation artefacts [22]. They developed ontologies to query the linking.

Lawrie *et al.* employed information retrieval techniques to measure how the comments relate to the respective source code and assume that comments impact the code quality of software systems [14]. Marcus and Poshyvanyk defined metrics for measuring the conceptual cohesion of classes [17]. For that they incorporated the presence (absence) of comments.

In [24], Ying *et al.* investigated the usage of a particular type of comment, the Eclipse task comments, *i.e.*, special comments starting with `// TODO` which are commonly used by developers using the Eclipse IDE. They argued that task comments tend to depend a lot on the context of the surrounding code and that it is difficult to infer the scope of a task comment. This often holds for comments in general and has therefore an impact on our work. Ying *et al.* mentioned a few reasons that lead to an insert of a comment task (for example as pointers to change requests) but they did not study whether some building blocks of a program (*e.g.*, `if`-statement) are more likely to be commented. Again, they did not analyze any evolutionary aspects either such as source code or comment changes.

Recently, some work was done on finding changes between different versions of a program. Xing and Stroulia presented an approach for detecting structural changes between the designs of subsequent versions of object-oriented software [23]. In [2], Apiwattanapong *et al.* detected changes in object-oriented programs based on differencing

of enhanced control flow graphs. Canfora *et al.* reconstruct changes from differencing results provided by CVS or Subversion *diff* [3]. In [15] Maletic and Collard present a language independent approach for detecting syntactic differences between source files—including comments—using an intermediate representation of the source code in XML. The output provided by *GNU diff* is mapped to an XML representation to locate changed entities. However, as far as we know, except for Maletic and Collard, none of the existing work in this area incorporates comments.

5. Conclusions and Future Work

In this paper, we investigated the co-evolution relation between source code and its associated comments. We presented a technique to map source code entities to comments in the code and to extract comment changes over the history of a software project. We have conducted three experiments on three different open source software systems to answer our research questions: ArgoUML, Azureus, and JDT Core.

Our findings showed that: When code and comments co-evolve, both are changed in the same revision: 97% of comment changes are done in the same revision as the associated source code change. But code and comments rarely co-evolve: despite its growth rate, newly added code barely is commented. And when it is commented, mostly class and method declarations are the target software entity.

For future work, we want to refine the comment mapping and extracting algorithm and also filter source code marked as comments. Still, we expect no surprising differences in the results. Further improvements will incorporate the scope of comments, *i.e.*, comments that describe more than a single statement, and an appropriate handling of commented source code. Since we have addressed open source software only for this study, we plan to also analyse commercial software to allow the findings to be compared. We plan to use results from this investigation to support developers in their daily tasks, for instance, by identifying the source code structures that are worth commenting or when comments should be updated.

Acknowledgments

This work was supported by the Swiss National Science Foundation as part of the COSE project, and the Hasler Foundation as part of the ProMedServices project.

References

[1] G. Antoniol, G. Canfora, G. Casazza, A. De Lucia, and E. Merlo. Recovering traceability links between code and documentation. *IEEE Trans. Software Eng.*, 28(10):970–983, 2002.

[2] T. Apiwattanapong, A. Orso, and M. J. Harrold. A differencing algorithm for object-oriented programs. In *Proc. Int'l Conf. Automated Software Eng.*, pages 2–13, 2004.

[3] G. Canfora, L. Cerulo, and M. D. Penta. Identifying changed source code lines from version repositories. In *Proc. Int'l Workshop Mining Software Repositories*, pages 14–14, 2007.

[4] S. S. Chawathe, A. Rajaraman, H. Garcia-Molina, and J. Widom. Change detection in hierarchically structured information. In *Proc. Int'l Conf. Management of Data*, pages 493–504, 1996.

[5] S. Demeyer, S. Ducasse, and O. Nierstraz. *Object-Oriented Reengineering Patterns*. Morgan Kaufmann, San Francisco, CA, USA, 2003.

[6] J. L. Elshoff and M. Marcotty. Improving computer program readability to aid modification. *Communications of the ACM*, 25(8):512–521, 1982.

[7] M. Fischer, M. Pinzger, and H. C. Gall. Populating a release history database from version control and bug tracking systems. In *Proc. Int'l Conf. Software Maintenance*, pages 23–32, 2003.

[8] B. Fluri and H. C. Gall. Classifying change types for qualifying change couplings. In *Proc. Int'l Conf. Program Comprehension*, pages 35–45, Athen, Greece, June 2006. IEEE Computer Society Press.

[9] B. Fluri, M. Würsch, M. Pinzger, and H. C. Gall. Change distilling—Tree differencing for fine-grained source code change extraction. *IEEE Trans. Software Eng.*, PrePrint, accepted for publication(to appear):17, July 2007.

[10] A. Goldberg. Programmer as reader. *IEEE Software*, 4(5):62–70, 1987.

[11] Z. M. Jiang and A. E. Hassan. Examining the evolution of code comments in postgresql. In *Proc. Int'l Workshop Mining Software Repositories*, pages 179–180, 2006.

[12] M. J. Kaelbling. Programming languages should NOT have comment statements. *SIGPlan Notices*, 23(10):59–60, 1988.

[13] A. Lakhotia. Understanding someone else's code: Analysis and experience. *Journal of Systems and Software*, 23(3):269–275, 2003.

[14] D. J. Lawrie, H. Feild, and D. Binkley. Leveraged quality assessment using information retrieval techniques. In *Proc. Int'l Conf. Program Comprehension*, June 2006.

[15] J. I. Maletic and M. L. Collard. Supporting source code difference analysis. In *Proc. Int'l Conf. Software Maintenance*, pages 210 – 219, September 2004.

[16] A. Marcus and J. I. Maletic. Recovering documentation-to-source-code traceability links using latent semantic indexing. In *Proc. Int'l Conf. Software Eng.*, pages 125–135, 2003.

[17] A. Marcus and D. Poshyvanyk. The conceptual cohesion of classes. In *Proc. Int'l Conf. Software Maintenance*, September 2005.

[18] D. Spinellis. *Code Quality—The Open Source Perspective*. Addison-Wesley, 2006.

[19] T. Tenny. Program readability: Procedures versus comments. *IEEE Trans. Software Eng.*, 14(9):1271–1279, 1988.

[20] M. L. V. D. Vanter. The documentary structure of source code. *Information and Software Technology*, 44(13):767–782, October 2002.

[21] A. Vermeulen, S. W. Ambler, G. Bumgardner, E. Metz, T. Misfeldt, J. Shur, and P. Thompson. *The Elements of Java Style*. Cambridge University Press, 2000.

[22] R. Witte, Y. Zhang, and J. Rilling. Empowering software maintainers with semantic web technologies. In *European Semantic Web Conf.*, pages 37–52, 2007.

[23] Z. Xing and E. Stroulia. Umldiff: an algorithm for object-oriented design differencing. In *Proc. Int'l Conf. Automated Software Eng.*, pages 54–65, 2005.

[24] A. T. T. Ying, J. L. Wright, and S. Abrams. Source code that talks: an exploration of eclipse task comments and their implication to repository mining. In *Proc. Int'l Workshop Mining Software Repositories*, pages 1–5, 2005.